KNOWNSEC

Degree of secrecy ★ | Project team open

# Smart Contract Audit Report

Security status

## Safe

★ ★ ★ ★ ★

Principal tester: Knownsec blockchain security team

# Version Summary

| Content | Date | Version |
| --- | --- | --- |
| Editing Document | 20210322 | V1.0 |

# Report Information

| Title | Version | Document Number | Type |
| --- | --- | --- | --- |
| **YouswapInviteV1 Smart Contract Audit Report** | V1.0 | | Open to project team |

# Copyright Notice

# Table of Contents

# 1. Introduction

The effective test time of this report is from From March 15, 2021 to March 15, 2021 . During this period, the security and standardization of **the smart contract code of the YouswapInviteV1** will be audited and used as the statistical basis for the report.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). **The smart contract code of the YouswapInviteV1** is comprehensively assessed as **SAFE**.

**Results of this smart contract security audit： SAFE**

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

**Report information of this audit:**

**Report Number：**

**Report query address link:**

**Target information of the YouswapInviteV1 audit:**

| Target information | |
|---|---|
| Token name | YouswapInviteV1 |
| Contract address | 0x25310873E310b270aEc5113A2d3037FA94166969 |
| Code type | Invitation relationship contract code, Ethereum smart contract code |
| Code language | solidity |

**Contract documents and hash:**

| Contract documents | MD5 |
|---|---|

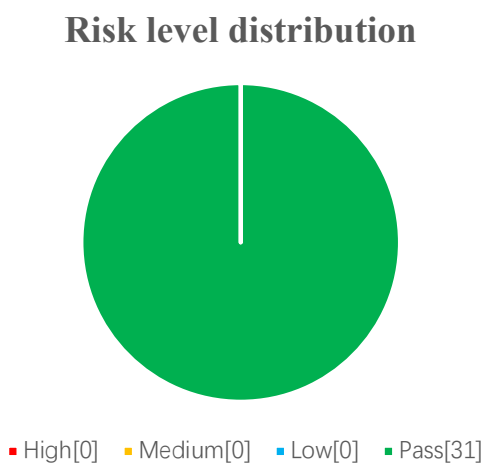
| | |
|---|---|
| YouswapInviteV1.sol | 44c2c4dd5ec1d15208166a78a091a237 |

# 2. Code vulnerability analysis

## 2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level：

| Vulnerability risk level statistics table | | | |
|---|---|---|---|
| High | Medium | Low | Pass |
| 0 | 0 | 0 | 31 |



## 2.2 Audit Result

| Result of audit | | | |
|---|---|---|---|
| **Audit Target** | **Audit** | **Status** | **Audit Description** |
| **Business security testing** | **YouswapInviteV1 contract variables and constructor** | Pass | After testing, there is no such safety vulnerability. |
| | **YouswapInviteV1 contract register function** | Pass | After testing, there is no such safety vulnerability. |
| | **YouswapInviteV1 contract acceptInvitation function** | Pass | After testing, there is no such safety vulnerability. |
| | **YouswapInviteV1 contract inviteBatch function** | Pass | After testing, there is no such safety vulnerability. |
| **Basic code vulnerability detection** | **Compiler version security** | Pass | After testing, there is no such safety vulnerability. |
| | **Redundant code** | Pass | After testing, there is no such safety vulnerability. |
| | **Use of safe arithmetic library** | Pass | After testing, there is no such safety vulnerability. |
| | **Not recommended encoding** | Pass | After testing, there is no such safety vulnerability. |
| | **Reasonable use of require/assert** | Pass | After testing, there is no such safety vulnerability. |
| | **fallback function safety** | Pass | After testing, there is no such safety vulnerability. |

| | | | |
|---|---|---|---|
| | **tx.oriigin authentication** | Pass | After testing, there is no such safety vulnerability. |
| | **Owner permission control** | Pass | After testing, there is no such safety vulnerability. |
| | **Gas consumption detection** | Pass | After testing, there is no such safety vulnerability. |
| | **call injection attack** | Pass | After testing, there is no such safety vulnerability. |
| | **Low-level function safety** | Pass | After testing, there is no such safety vulnerability. |
| | **Vulnerability of additional token issuance** | Pass | After testing, there is no such safety vulnerability. |
| | **Access control defect detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Numerical overflow detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Arithmetic accuracy error** | Pass | After testing, there is no such safety vulnerability. |
| | **Wrong use of random number detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Unsafe interface use** | Pass | After testing, there is no such safety vulnerability. |
| | **Variable coverage** | Pass | After testing, there is no such safety vulnerability. |
| | **Uninitialized storage pointer** | Pass | After testing, there is no such safety vulnerability. |

| | | | |
|---|---|---|---|
| | **Return value call verification** | Pass | After testing, there is no such safety vulnerability. |
| | **Transaction order dependency detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Timestamp dependent attack** | Pass | After testing, there is no such safety vulnerability. |
| | **Denial of service attack detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Fake recharge vulnerability detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Reentry attack detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Replay attack detection** | Pass | After testing, there is no such safety vulnerability. |
| | **Rearrangement attack detection** | Pass | After testing, there is no such safety vulnerability. |

# 3. Analysis of code audit results

## 3.1. YouswapInviteV1 contract variables and constructor 【PASS】

**Audit analysis:** YouswapInviteV1 contract variable definition and the design of the constructor are reasonable. There are unused variables in the error message library used in the contract, it is recommended to delete to streamline the code.

```
/**
 *Submitted for verification at Etherscan.io on 2021-03-17
*/

// SPDX-License-Identifier: MIT

pragma solidity 0.7.4;

library ErrorCode {
    //knownsec// It is recommended to delete the unused error message to explain the variable declaration

    string constant FORBIDDEN = 'YouSwap:FORBIDDEN';
    string constant IDENTICAL_ADDRESSES = 'YouSwap:IDENTICAL_ADDRESSES';
    string constant ZERO_ADDRESS = 'YouSwap:ZERO_ADDRESS';
    string constant INVALID_ADDRESSES = 'YouSwap:INVALID_ADDRESSES';
    string constant BALANCE_INSUFFICIENT = 'YouSwap:BALANCE_INSUFFICIENT';
    string constant REWARDTOTAL_LESS_THAN_REWARDPROVIDE = 'YouSwap:REWARDTOTAL_LESS_THAN_REWARDPROVIDE';
    string constant PARAMETER_TOO_LONG = 'YouSwap:PARAMETER_TOO_LONG';
    string constant REGISTERED = 'YouSwap:REGISTERED';

}
```

```
interface IYouswapInviteV1 {

    struct UserInfo {
        address upper;//上级
        address[] lowers;// 下级
        uint256 startBlock;//邀请块高
    }

    event InviteV1(address indexed owner, address indexed upper, uint256 indexed height);//被邀请人的地址，邀请人的地址，邀请块高

    function inviteCount() external view returns (uint256);//邀请人数

    function inviteUpper1(address) external view returns (address);//上级邀请

    function inviteUpper2(address) external view returns (address, address);//上级邀请

    function inviteLower1(address) external view returns (address[] memory);// 下级邀请

    function inviteLower2(address) external view returns (address[] memory, address[] memory);// 下级邀请

    function inviteLower2Count(address) external view returns (uint256, uint256);// 下级邀请

    function register() external returns (bool);//注册邀请关系

    function acceptInvitation(address) external returns (bool);//注册邀请关系

    function inviteBatch(address[] memory) external returns (uint, uint);//注册邀请关系：输入数量，成功数量

}
```

```
contract YouswapInviteV1 is IYouswapInviteV1 {

    address public constant ZERO = address(0);
    uint256 public startBlock;
    address[] public inviteUserInfoV1;
    mapping(address => UserInfo) public inviteUserInfoV2;

    constructor () {
        startBlock = block.number;
        }
```

**Recommendation：** nothing.

## 3.2. YouswapInviteV1 contract register function【PASS】

**Audit analysis:** The register function of the YouswapInviteV1 contract is used to register user information.

```
  //knownsec// registered
    function register() override external returns (bool) {
        UserInfo storage user = inviteUserInfoV2[tx.origin]; //knownsec// Read the inviter information of the transaction initiator, restrict the inviter to the contract address
        require(0 == user.startBlock, ErrorCode.REGISTERED); //knownsec// Unregistered user check
        user.upper = ZERO; //knownsec// Update inviter superior
        user.startBlock = block.number; //knownsec// Update the starting block number of the registered inviter
        inviteUserInfoV1.push(tx.origin); //knownsec// Write the transaction initiation address to the inviter list

        emit InviteV1(tx.origin, user.upper, user.startBlock);

        return true;
    }
```

**Recommendation：** nothing.

## 3.3. YouswapInviteV1 contract acceptInvitation function 【PASS】

**Audit analysis:** The acceptInvitation function of the YouswapInviteV1 contract is used to receive an invitation from a specified address, and set the specified address as its superior relationship. If the superior invite address is not registered, the registered user information will be automatically added to the superior address.

```
    //knownsec// Accept invitation from superior
    function acceptInvitation(address _inviter) override external returns (bool) {
        require(msg.sender != _inviter, ErrorCode.FORBIDDEN); //knownsec// Not allowed to invite yourself
        UserInfo storage user = inviteUserInfoV2[msg.sender]; //knownsec// Read the user information of the invitee
        require(0 == user.startBlock, ErrorCode.REGISTERED); //knownsec// Unregistered user check
        UserInfo storage upper = inviteUserInfoV2[_inviter]; //knownsec// Read the user information of the inviter
        if (0 == upper.startBlock) {
            upper.upper = ZERO; //knownsec// If the inviter does not register, he will record his own superior address as zero
            //knownsec// registered user
            upper.startBlock = block.number;
            inviteUserInfoV1.push(_inviter);

            emit InviteV1(_inviter, upper.upper, upper.startBlock);
        }
        user.upper = _inviter; //knownsec// Record your own superior as the inviter's address
        upper.lowers.push(msg.sender); //knownsec// Save the caller to the lower-level address
```

```
list of the inviter
            //knownsec// registered user
            user.startBlock = block.number;
            inviteUserInfoV1.push(msg.sender);

            emit InviteV1(msg.sender, user.upper, user.startBlock);

            return true;
        }
```

**Recommendation：** nothing.

## 3.4. YouswapInviteV1 contract inviteBatch function【PASS】

**Audit analysis:** The inviteBatch function of the YouswapInviteV1 contract is used to invite subordinates in batches.

```
    //knownsec// Bulk invite subordinates
    function inviteBatch(address[] memory _invitees) override external returns (uint, uint) {
        uint len = _invitees.length;
        require(len <= 100, ErrorCode.PARAMETER_TOO_LONG); //knownsec// Limit on the number of batch invitations
        UserInfo storage user = inviteUserInfoV2[msg.sender]; //knownsec// Read caller user information
        //knownsec// Update registration information for unregistered users
        if (0 == user.startBlock) {
            user.upper = ZERO;
            user.startBlock = block.number;
            inviteUserInfoV1.push(msg.sender);

            emit InviteV1(msg.sender, user.upper, user.startBlock);
        }
        uint count = 0;
        for (uint i = 0; i < len; i++) { //knownsec// Traverse setting invitation relationship
```

```
                if ((address(0) != _invitees[i]) && (msg.sender != _invitees[i])) { //knownsec// Don't invite yourself
                    UserInfo storage lower = inviteUserInfoV2[_invitees[i]]; //knownsec// Read the user information of the i-th invitee
                    if (0 == lower.startBlock) {
                        lower.upper = msg.sender; //knownsec// Set up superior relationship
                        lower.startBlock = block.number; //knownsec// Set registration block number
                        user.lowers.push(_invitees[i]); //knownsec// Store the lower-level list of invitees
                        inviteUserInfoV1.push(_invitees[i]); //knownsec// Store the list of registered users
                        count++;

                        emit InviteV1(_invitees[i], msg.sender, lower.startBlock);
                    }
                }
            }

            return (len, count);
        }
```

**Recommendation：** nothing.

# 4. Basic code vulnerability detection

## 4.1. Compiler version security【PASS】

Check whether a safe compiler version is used in the contract code implementation.

**Audit result:** After testing, the smart contract code has formulated the compiler version 0.6.0 within the major version, and there is no such security problem.

**Recommendation：** nothing.

## 4.2. Redundant code【PASS】

Check whether the contract code implementation contains redundant code.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.3. Use of safe arithmetic library【PASS】

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

**Audit result:** After testing, the SafeMath safe arithmetic library has been used in the smart contract code, and there is no such security problem.

**Recommendation：** nothing.

## 4.4. Not recommended encoding【PASS】

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.5. Reasonable use of require/assert【PASS】

Check the rationality of the use of require and assert statements in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.6. Fallback function safety【PASS】

Check whether the fallback function is used correctly in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.7. tx.origin authentication【PASS】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.8. Owner permission control【PASS】

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.9. Gas consumption detection【PASS】

Check whether the consumption of gas exceeds the maximum block limit.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.10. call injection attack【PASS】

When the call function is called, strict permission control should be done, or the function called by the call should be written dead.

**Audit result:** After testing, the smart contract does not use the call function, and this vulnerability does not exist.

**Recommendation：** nothing.

## 4.11. Low-level function safety【PASS】

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.12. Vulnerability of additional token issuance【PASS】

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

**Audit result:** After testing, the smart contract code does not have the function of issuing additional tokens, and the upper limit is set, so it is passed.

**Recommendation：** nothing.

## 4.13. Access control defect detection【PASS】

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.14. Numerical overflow detection【PASS】

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers (2^256-1). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect

results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.15. Arithmetic accuracy error【PASS】

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations: 5/2*10=20, and 5*10/2=25, resulting in errors, which are larger in data The error will be larger and more obvious.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.16. Incorrect use of random numbers【PASS】

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as block.number and block.timestamp, they are usually more public than they appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.17. Unsafe interface usage【PASS】

Check whether unsafe interfaces are used in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.18. Variable coverage【PASS】

Check whether there are security issues caused by variable coverage in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.19. Uninitialized storage pointer【PASS】

In solidity, a special data structure is allowed to be a struct structure, and the local variables in the function are stored in storage or memory by default.

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

**Audit result:** After testing, the smart contract code does not use structure, there is no such problem.

**Recommendation：** nothing.

## 4.20. Return value call verification【PASS】

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send ETH to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be

returned when call.value fails to be sent; all available gas will be passed for calling (can be Limit by passing in gas_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to ETH sending failure.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.21. Transaction order dependency【PASS】

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

**Audit result:** After testing, the _approve function in the contract has a transaction sequence dependency attack risk, but the vulnerability is extremely difficult to exploit, so it is rated as passed. The code is as follows:

```
function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");
```

```
        _allowances[owner][spender] = amount;//knownnsec// Transaction order depends on
risk        emit Approval(owner, spender, amount);
    }
```

**The possible security risks are described as follows:**

1. By calling the approve function, user A allows user B to transfer money on his behalf to N (N>0);

2. After a period of time, user A decides to change N to M (M>0), so call the approve function again;

3. User B quickly calls the transferFrom function to transfer N number of tokens before the second call is processed by the miner;

4. After user A's second call to approve is successful, user B can obtain M's transfer quota again, that is, user B obtains N+M's transfer quota through the transaction sequence attack.

**Recommendation：**

1. Front-end restriction, when user A changes the quota from N to M, he can first change from N to 0, and then from 0 to M.

2. Add the following code at the beginning of the approve function:

require((_value == 0) || (allowed[msg.sender][_spender] == 0));

## 4.22. Timestamp dependency attack【PASS】

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block

and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.23. Denial of service attack【PASS】

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state. There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.24. Fake recharge vulnerability【PASS】

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.25. Reentry attack detection【PASS】

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send ETH. When the **call.value()** function to send ETH occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

## 4.26. Replay attack detection【PASS】

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

In the asset management system, there are often cases of entrusted management. The principal assigns assets to the trustee for management, and the principal pays a certain fee to the trustee. This business scenario is also common in smart contracts.

**Audit results：** After testing, the smart contract does not use the call function, and this vulnerability does not exist.

**Recommendation：** nothing.

## 4.27. Rearrangement attack detection【PASS】

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

**Audit results：** After auditing, the vulnerability does not exist in the smart contract code.

**Recommendation：** nothing.

# 5. Appendix A：Contract code

**Source code：**

*YouswapInviteV1*

*https://etherscan.io/address/0x25310873e310b270aec5113a2d3037fa94166969#code*

# 6. Appendix B：Vulnerability rating standard

| *Smart contract vulnerability rating standards* | |
|---|---|
| Level | Level Description |
| High | Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.;<br>Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.;<br>Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of gas. |
| Medium | High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc. |
| Low | Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of ETH or tokens to trigger, vulnerabilities where attackers cannot |

| | |
|---|---|
| | directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk Wait. |

# 7. Appendix C：Introduction to auditing tools

## 7.1 Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

## 7.2 Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

## 7.3 securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a

specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

## 7.4 Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

## 7.5 MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

## 7.6 ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

## 7.7 ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 7.8 Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 7.9 Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

KNOWNSEC

Beijing KnownSec Information Technology Co., Ltd.

| | |
|---|---|
| Advisory telephone | +86(10)400 060 9587 |
| E-mail | sec@knownsec.com |
| Website | www.knownsec.com |
| Address | wangjing soho T2-B2509,Chaoyang District, Beijing |